Shortest Path Routing on the Hypercube with Faulty Nodes

Mehrdad Arabpour Niasari

Department of Computer Science

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

This thesis is dedicated to my parents
for their love and endless support.

Abstract

Interconnection networks are widely used in parallel computers. There are many topologies for interconnection networks and the hypercube is one of the most popular networks. There are a variety of different routing paradigms that need to be investigated on the hypercube. In this thesis we investigate the shortest path routing between two nodes on the hypercube when some nodes are faulty and cannot be used. In this thesis the shortest path between two nodes is considered as the Hamming distance of them.

Regarding the shortest path problem in a faulty hypercube, some efficient algorithms have been proposed when each processor (node) has limited information regarding the status of other processors (whether they are faulty or not). There are also some proposed algorithms for the case where there is no limitation on the data of each processor but they are not efficient and are exponential in terms of number of faulty nodes and dimension of the hypercube.

To check whether there is a shortest path between two given nodes in a faulty hypercube, we propose a polynomial algorithm with time complexity of $O(n^2m^2)$ where n is the dimension of the hypercube and m is the number of faulty nodes. Our algorithm only requires the source node to know the state of all other nodes. The proposed algorithm first checks whether there is a shortest path from the source node to the target node and then it can construct it efficiently.

Our idea is based on a so-called ordering and permutation model of paths in the hypercube. We use a constructive approach to find the path which is a permutation as well. We then use inclusion-exclusion and dynamic programming techniques to make our method efficient. We also propose an algorithm for counting all possible shortest paths in the hypercube.

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Dr. Ke Qiu, for his continuous support, patience, enthusiasm and motivation throughout this process. His guidance has helped me in all the time of research and without him this thesis would not have been completed or written.

Besides my advisor, I would like to thank the members of my supervisory committee, Dr. S. Houghten, Dr. M. Winter, and Dr. B. Farzad, for their time and insightful comments.

Last but not the least, I must express my very profound gratitude to my loving family, my parents and brother for their unconditional and invaluable love and support and my charming little sister for helping me to be spiritually up. No matter where they are around the world, they are always in my heart. This accomplishment would not have been possible without them. Thank you.

M.A

Contents

List of Tables

List of Figures

Chapter 1

Introduction to Parallel Computers

1.1 Introduction

Computers can be categorized in two major types based on the number of processors they use. Computers with a single processor are called sequential computers and the ones with more than one processor are considered as parallel computers.

In sequential computers, a program is defined as a sequence of instructions which tells the processor how and in which order to solve a certain problem. In this architecture there are five main units, namely: input, control, memory, processor and output units. Briefly, the control unit obtains an instruction from the memory unit and passes it to the processor for doing a certain arithmetic or logical operation. The processor is also connected to input and output units to be able to communicate with the outside world. Also the processor has a local memory to perform its computations.

In parallel computers, by contrast, there is more than one processor and by breaking the input problem into smaller subproblems, each processor is assigned to solve one of them. The processors may also communicate with each other to exchange the partial result to obtain the final answer of the original problem.

There are many different classifications of parallel computers. The one that we discuss here is based on their communication medium, whether it is a shared memory or an interconnection network. In this chapter we first go over a general classification of computer architectures. Then we introduce two main computational models, shared memory and interconnection networks. We focus more on different instances and properties of the latter network since in this thesis we propose an algorithm for one important interconnection network, the hypercube. We also review two variants of the hypercube which can be a part of future work on the proposed algorithm.

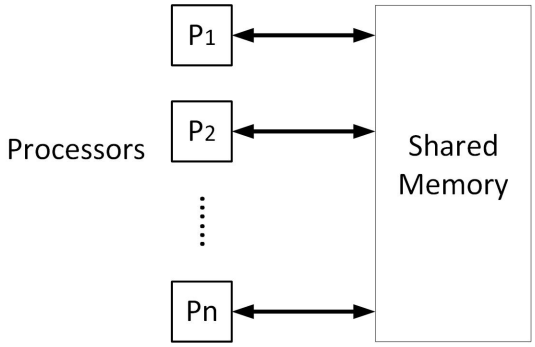

Figure 1.1: A shared memory parallel computer

1.2 Shared Memory Parallel Computers

In shared memory parallel computers, as shown in Figure 1.1, multiple processors operate independently but share a single and common memory. They communicate to each other through the shared resource and every change of one processor is visible to all other processors. This model is also known as Parallel Random Access Machine (PRAM) since the machine is able to have access to any unit of data randomly.

PRAM computers are classified into four submodels based on the way the processors gain access to the shared memory.

- **Exclusive Read, Exclusive Write (EREW)**: in which no two processors should have access to the same location of shared memory at the same time.
- **Exclusive Read, Concurrent Write (ERCW):** in which multiple processors can write to but not read from the same location of memory at the same time.
- **Concurrent Read, Exclusive Write (CREW):** in which multiple processors can read from but not write to the same location of memory simultaneously.
- **Concurrent Read, Concurrent Write (CRCW):** in which multiple processors can either write to or read from the same location.

1.3 Interconnection Networks

Interconnection networks are another architecture for parallel computers that is generally designed for fast and reliable communication among the processors [11]. Despite shared memory architecture, there is no shared memory involved in interconnection networks and each processor has its own dedicated memory. In order for processors to

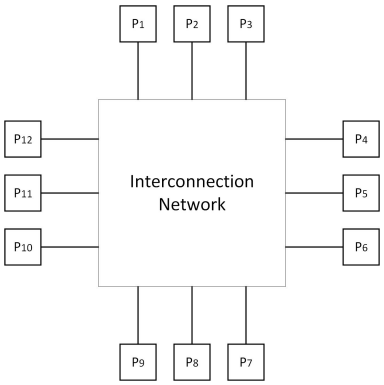

Figure 1.2: Interconnection network

communicate with each other there are different topologies that indicate in which way there are links between processors. Therefore, communication between processors is done by message passing. Also the links can be one way or two ways.

The entire architecture can be modeled with a graph $G = (V, E)$ in which V is a set of nodes (each processor is a node) and E is the set of links, i.e. $E \subseteq V \times V$. Thus, we will use some standard terminologies of graph theory in this thesis. Also the terms "processor" and "node" and "vertex", "edge" and "link", "interconnection network" and "graph" will be used interchangeably. Before discussing some important and well-known topologies, we start with reviewing definitions of some graph theory terminologies that we will use in the rest of the thesis.

Definition 1. A *path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $\forall i \in \{1, 2, \ldots, k-1\}$, v_i and v_{i+1} are adjacent, i.e. $(v_i, v_{i+1}) \in E$. A cycle is a closed path, i.e. $v_1 = v_k$.

Definition 2. The *distance* between two nodes of graph is defined as the number of edges connecting them in a shortest path. The *diameter* of graph is defined as the maximum pairwise distance between any of its vertices.

Definition 3. A directed acyclic graph (DAG) is a graph in which its edges are directed and the graph has no cycle.

Definition 4. An undirected graph is called a *tree*, if for every two vertices there is exactly one path connecting them. A rooted tree is an acyclic connected graph (i.e. all nodes are connected) with one node considered as the root of the tree. A tree can also be defined recursively as follows: a single node is a tree. If $T_1, T_2, \ldots, T_k$ are disjoint trees with roots $t_1, t_2, \ldots, t_k$, a graph that is formed by connecting a new vertex r to all roots r_i is a tree with root r. The roots $t_1, t_2, \ldots, t_k$ are called children of r and r is called the parent of r_i's.

Definition 5. A rooted tree is called a *binary tree* if each node of the tree has at most two children.

Definition 6. Graph H is *isomorphic* to graph G if there is a bijection $f : V(G) \to V(H)$ such that $(u, v) \in E(G)$ if and only if $(f(u), f(v)) \in E(H)$. An automorphism of graph G is defined as an isomorphism of G into G.

Definition 7. A graph is considered as node-symmetric (edge-symmetric) if for every pair of $u, v \in V(G)$ $(e, f \in E)$ there is an automorphism that maps u to v (e to f).

The latter definition makes it possible to look at the graph from any node and get exactly the same graph. Being symmetric is considered an important characteristic of a network since designing algorithms for routing and broadcasting is easier due to the same accessibility between the processors. For example, for a node-symmetric network, we can propose an algorithm based on a specific node and then it can be easily generalized for any other nodes. We have leveraged the symmetry property of the hypercube in our proposed method which will be discussed in detail.

There are some important metrics about interconnection networks that determine its performance and practical usability in real life applications. Some of these metrics are as follows:

- What is the longest shortest path in the network among all pairs of processors? In other words, in the worst case how long does it take for two processors to communicate with each other?
- How many neighbours does each processor have?
- How should a message communicate between two arbitrary processors?
- Is a specific processor able to communicate with a set of other processors via node-disjoint paths?
- If there are some nodes which are not accessible (faulty nodes) how flexible is the network for processors to continue communicating?

Since different parallel systems can have different usages, there are many topologies to cover each factor better than others. We review some well-known and important proposed topologies and their properties. For the remainder of the thesis, we use N as the number of processors and M as the number of links in each topology.

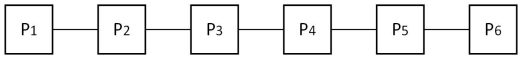

Figure 1.3: Linear array with 6 nodes

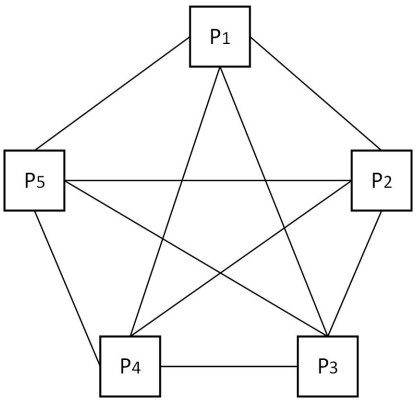

Figure 1.4: Complete network with 5 nodes

1.3.1 Linear Array and Ring

In a linear array, processors simply form a chain where each processor has two neighbours except for the first and last one, i.e. the corresponding graph has N vertices numbered from 1 to N and $(v_i, v_{i+1}), (v_{i+1}, v_i) \in E$ for each $1 \leq i < N - 1$. Figure 1.3 shows a linear array of size 6. The major advantage of this simple network is ease of implementation. The diameter is $N - 1$ that is rather long for large N. If the first and last nodes are connected to each other we will have a ring network. The main reason of having such a network is to reduce the diameter to $\lfloor N/2 \rfloor$ compared to the linear array. The routing algorithms in these architectures are easy but the average distance between nodes grows linearly with N which is not desirable in large networks.

1.3.2 Complete Graph

In the extreme case each processor can be directly connected to all other processors, i.e. $E = V \times V$. In this case, the diameter of the graph becomes 1. This is called a complete network and is the most dense network with the maximum possible links $\binom{N}{2}$. Figure 1.4 gives a complete graph of five nodes K_5. Clearly, it is impractical to build a complete network for large N.

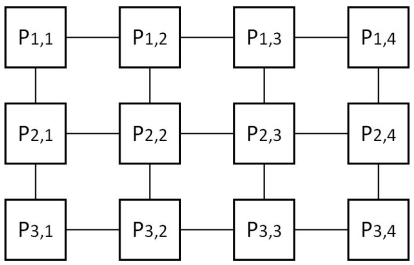

Figure 1.5: A mesh of size 3×4

1.3.3 Mesh and Torus

The mesh topology is an $m \times n$ grid with m rows and n columns. Each processor (except for the ones on the boundary of the grid) is connected to four other processors (top, left, right and bottom of its position in the grid). In the corresponding graph, nodes are numbered as a grid, i.e. $v_{i,j}$ and it is connected to $v_{i+1,j}, v_{i-1,j}, v_{i,j+1}, v_{i,j-1}$ if they exist (indices are within the range). It can be shown easily that the diameter of this network is $O(m+n)$ and also is a non-symmetric network. Since this topology is easy to lay out, it has been used widely in multiprocessor systems.

A torus network is similar to mesh with some extra edges that connect the first and last nodes of each row and each column to each other. This makes boundary nodes to have the same characteristics as internal nodes. Mesh and torus can also be generalized to higher dimensions. Figure 1.5 shows a mesh of size 3×4.

1.3.4 Tree

This network is a complete binary tree where each processor except for the root node and leaf nodes, is connected to its parent and two children and each level of tree is completely filled and in the last level all nodes are in the left side. Therefore, for a binary tree with d levels, there are $2^d - 1$ nodes. In other words, there would be $\lceil \log_2 N \rceil$ levels which is asymptotically equal to the diameter. A tree with 4 levels is shown in Figure 1.6.

1.3.5 Hypercube

Since the main focus on this thesis is on the hypercube, we discuss its important and related characteristics in detail.

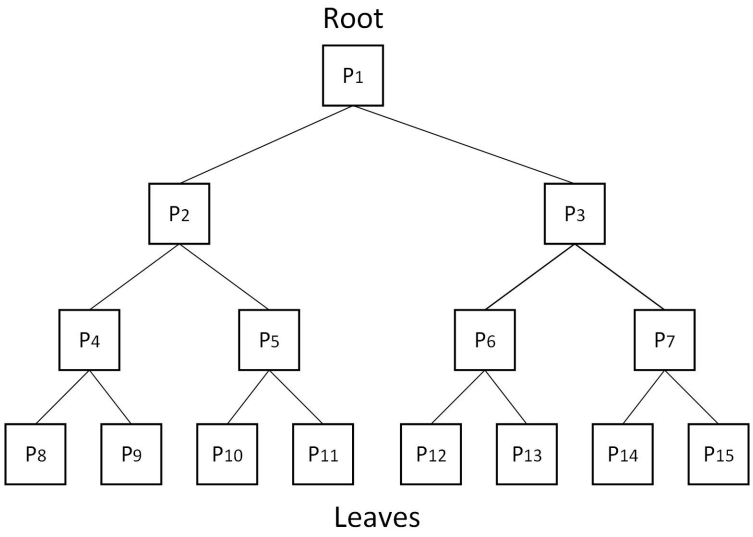

Figure 1.6: Tree with 4 levels and 15 nodes

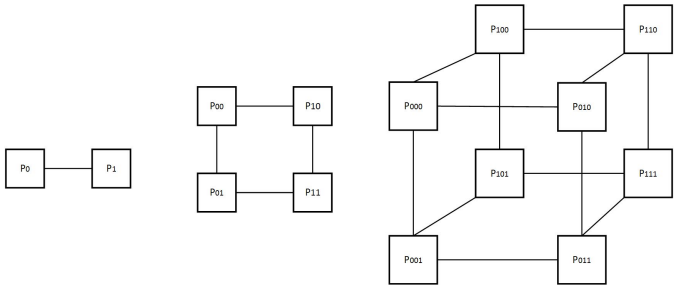

Figure 1.7: Hypercubes of dimension 1, 2 and 3

Definition

An $n-$dimensional hypercube, also called $n-$cube or Q_n, has 2^n nodes that are numbered from 0 to $2^n - 1$ and labeled by the binary representation of their numbers. There is an edge between two nodes if and only if the binary representation of their numbers (i.e. their labels) differ in exactly one bit. More formally node $u = u_0u_1 \dots u_{i-1}u_iu_{i+1} \dots u_{n-1}$ is connected to $v = u_0u_1 \dots u_{i-1}\overline{u_i}u_{i+1} \dots u_{n-1}$ for all $0 \leq i \leq n-1$ ($\overline{0} = 1$ and $\overline{1} = 0$). The examples of hypercubes of size 1, 2 and 3 are shown in Figure 1.7. The hypercube is considered as a highly concurrent loosely coupled multiprocessor based topology due to its properties.

Symmetric Structure

The hypercube is node and edge symmetric. This implies that for routing and some other algorithms, we can assume that the source node is 0^n. For instance, consider

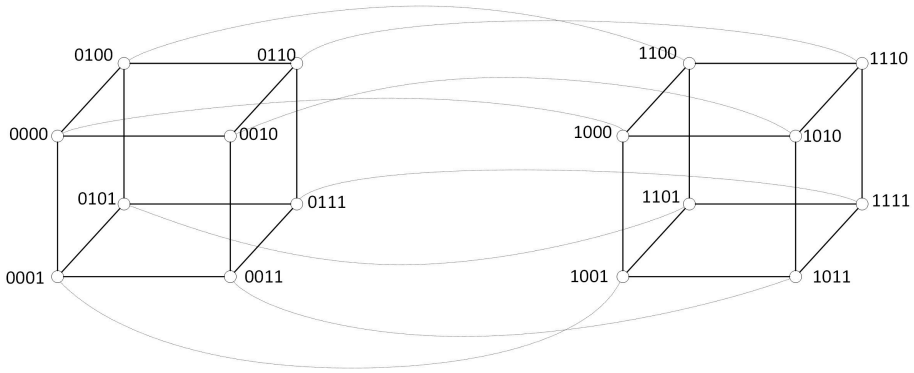

Figure 1.8: A 4-cube

the problem of finding shortest disjoint paths from given node s to a set of nodes $t_1, t_2, \ldots, t_k$. We can safely assume that $s = 0^n$ and present our algorithm. This is because all nodes are the same and we can relabel nodes (by an automorphism bijection) of the original hypercube in order to get a new hypercube such that given s is 0^n.

Recursive Structure

One of the most interesting and important properties of the hypercube is that it can be constructed recursively from cubes with lower dimensions. More precisely, an $n-$cube can be decomposed based on the value of the leading bit of its labels. Therefore, one subgraph will have all nodes whose leading bit is 0, and the other subgraph would have the remaining nodes, the ones with the leading bit of 1. The two subgraphs are such that each node of the first one is connected to one node of the other one. If we remove those interconnecting edges, we get two disjoint cubes that are isomorphic to $(n-1)-$cubes. This is illustrated in Figure 1.9 and Figure 1.8. The decomposition can be done based on any of the n bits, not necessarily on leading bit. Therefore, there are n different ways of decomposing an $n-$cube into two $(n-1)-$cubes.

This recursive structure makes it possible to run some recursive algorithms on hypercubes, such as divide-and-conquer or dynamic programming.

Shortest Path and Diameter

The Hamming distance of two nodes in Q_n is defined as the number of positions in their binary representations which are different. More formally, consider $u = u_0u_1 \ldots u_{n-1}$ and $v = v_0v_1 \ldots v_{n-1}$, the Hamming distance will be $H(u, v) = |\{i|u_i \neq$

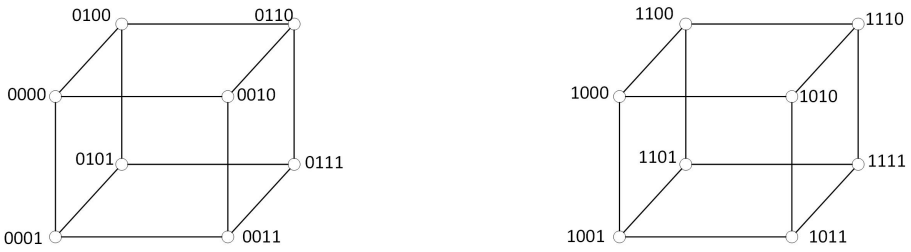

Figure 1.9: A decomposed 4-cube into two 3-cubes

$v_i\}|$. For instance, in Q_4 we have $H(6, 11) = H(0110, 1011) = |\{0, 1, 3\}| = 3$. The Hamming distance makes it easier to discuss about paths in the hypercube since two nodes are connected if their Hamming distance is 1. In Q_n, the shortest path between two nodes u and v is equal to $H(u, v)$. The approach is simply moving across the binary sequence of node v from left to right and flipping each bit that is different in the same position of u. Therefore, the total number of steps will be the same as $H(u, v)$.

The diameter of Q_n is n. It is simply because the maximum value of the Hamming distance among all pairs is at most n. Thus the longest shortest path or the diameter is n. Also note that since $N = 2^n$, Q_n has a logarithmic diameter which is an important property.

1.3.6 Routing Paradigms

There are many routing paradigms in the hypercube. In this thesis, our focus is on the shortest path paradigms. For example, given two nodes, how to find a shortest path between them? How many shortest paths exist in total? If the input is one node as the source node and a set of other nodes as the destination, how should we find the shortest path from source to all other nodes such that no two paths share a common node except the source (which is called disjoint paths)? Under which conditions do those shortest paths exist? What if some nodes become faulty and cannot be used, how can we check whether shortest paths exists between nodes and how to find them if exist?

This thesis propose an efficient algorithm to check whether there are disjoint shortest paths between one source node and a set of other nodes in the hypercube. We will define the problem and investigate it with details in Chapter 2 and Chapter 3.

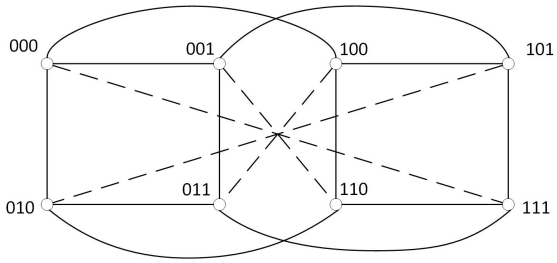

Figure 1.10: 3-dimensional folded-cube

1.3.7 Variants

Many cube-like networks have been proposed to have a better efficiency than hypercube in different aspects.

The *folded-cube* proposed by [8] is obtained by adding extra edges to the standard hypercube. It defines an edge from each node to the complement of its binary representation in the hypercube. New optimal routing algorithms have been developed for this cube-like network which are remarkably more efficient than the conventional hypercube [8] . The diameter is also reduced to $\lceil n/2 \rceil$.

Another interesting variance similar to the hypercube is called the *augmented cube* [6]. Augmented cube is also a node-symmetric cube and has diameter $\lceil n/2 \rceil$. The augmented cube has been defined recursively as follows:

Definition 8. The augmented cube AQ_n of dimension n has 2^n vertices, each labeled by an n-bit binary string $a_0a_1 \dots a_{n-1}$. We define $AQ_1 = K_2$ (a complete graph of size 2). For $n \geq 2, AQ_n$ is obtained by taking two copies of the augmented cube AQ_{n-1}, denoted by AQ^0_{n-1} and AQ^1_{n-1} , and adding $2 \times 2^{n-1}$ edges between the two as follows:

Let:

$V(AQ^0_{n-1}) = \{0a_1a_2 \dots a_{n-1} : a_i = 0 \ or \ 1\}$ and

$V(AQ^1_{n-1}) = \{1b_1b_2 \dots b_{n-1} : b_i = 0 \ or \ 1\}$

A vertex $A = 0a_1a_2 \dots a_{n-1}$ of AQ^0_{n-1} is joined to a vertex $B = 1b_1b_2 \dots b_{n-1}$ of AQ^1_{n-1} iff for every $i, 1 \leq i \leq n-1$, either

(i) $a_i = b_i$; in this case, AB is called a hypercube edge, or

(ii) $a_i = \overline{b_i}$; in this case, AB is called a complement edge [6].

There are many other cube-like networks such as the *exchanged hypercube* [15], the *twisted cube* [17], the *Fibonacci cube* [10], *cube connected cycles* [16], *dual cube* [14] etc. We defined the folded-cube and augmented cube since we believe due to

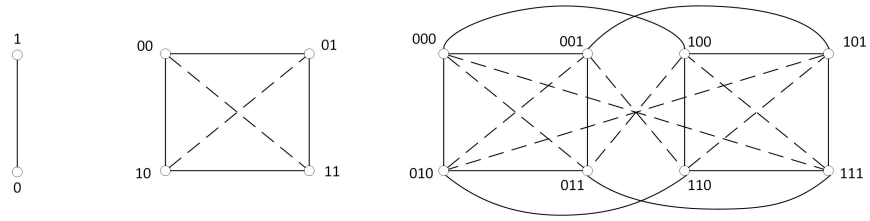

Figure 1.11: Augmented cubes of dimension 1, 2 and 3

their specific similarity with the hypercube, the proposed algorithm in this thesis can be examined on them as future work.

1.4 Organization of the Thesis

We discussed some important routing paradigms that need to be examined on every interconnection networks. We then focused on definition of the hypercube as an important and popular interconnection network and reviewed its properties. In this thesis, we examine the problem of finding the shortest path between two given nodes in the hypercube when some nodes are faulty nodes and cannot be used. Therefore, in Chapter 2 we review previous results on some sufficient conditions and algorithms for the problem. In Chapter 3 we propose our efficient algorithm and its proof. In Chapter 4 we talk about future work that can be done on the proposed algorithm.

Chapter 2

Literature Review

2.1 Introduction

The proposed algorithm in this thesis is about the problem of determining the existence of shortest path between two given nodes in the presence of some blocking nodes in a hypercube and finding if such a path exists. Note that as mentioned in Chapter 1, the length of a shortest path between two nodes u and v in the hypercube is equal to their Hamming distance, $H(u, v)$. We define the problem more precisely in the next chapter. In this chapter, we review some studies and proposed algorithms related to the same shortest path problem and similar routing paradigms in the hypercube.

2.2 Importance of Routing Paradigms

In multiprocessor systems, processors should be able to exchange data with each other via the links. This can be more challenging and problematic since there is no shared memory and data should travel along a path node by node. By definition, the length of a path is simply the number of edges on the path. There are many routing paradigms involved here, such as:

- How to find a (shortest) path between any two nodes?
- Assume two sets of nodes need to communicate at the same time, are there node-disjoint paths between the two sets? How to find them?
- If some processors are not working, how to find a (shortest) path between the two nodes without using those faulty nodes?

- How to find disjoint paths between two sets of nodes such that the summation of the lengths of the paths are shortest?

We review some of the theorems and algorithms that have been proposed for routing paradigms on the hypercube in the following sections.

2.3 Node-disjoint Shortest Path

In Q_n, assume v, as a source node and $v_1, v_2, \ldots, v_s$ any other nodes, necessarily distinct, as target nodes with $s \leq n$. The problem of finding necessary and sufficient conditions under which there are internally node-disjoint shortest paths from v to all other $v_1, v_2, \ldots, v_s$ has been studied by Gao, Qiu, et al [9]. They proposed a necessary and sufficient condition using Hall's matching theorem. They introduced a modeling called *disjoint ordering* that is the basis of their algorithm. Since the proposed method of this thesis is also inspired by disjoint ordering, we review the proposed conditions and modeling in detail.

Mathematical definition of disjoint ordering as discussed in [9] is as follows:

Definition 9. A permutation of the elements of a finite set is called an ordering. Let X and Y be two sets ordered as $O_X = (x_1, x_2, \ldots, x_a)$ and $O_Y = (y_1, y_2, \ldots, y_b)$, we say that O_X and O_Y are disjoint if

$$\{x_1, x_2, \ldots, x_i\} \neq \{y_1, y_2, \ldots, y_i\}$$

for $1 \leq i \leq min(|X|, |Y|)$ unless $i = |X| = |Y|$.

For instance, if $X = Y = \{1, 2, 3, 4\}$ then $O_X = (1, 3, 2, 4)$ and $O_Y = (2, 3, 4, 1)$ are disjoint while $O_X = (2, 3, 1, 4)$ and $O_Y = (1, 3, 2, 4)$ are not, since the set of the first three elements are equal, i.e. $\{1, 3, 2\} = \{2, 3, 1\}$.

The mentioned node-disjoint shortest paths problem can be modeled as a disjoint ordering problem. Before reviewing their modeling, we discuss how paths in the hypercube can be represented using sets and orderings.

As discussed in Chapter 1, the length of a shortest path between two nodes u and v in the hypercube is equal to $H(u, v)$. Also for constructing the shortest path, all we need is to flip the bits in the binary representation of u which are different from the binary representation of v. This can also be modeled as sets and ordering. If we make a set out of all bits that need to be changed, then any ordering of the set is a different shortest path. For instance, let $u = 01100$ and $v = 10101$ two nodes in Q_5.

Assume bits are numbered from left to right, starting with one. Therefore we have to flip bits $1, 2, 5$ in any arbitrary order to get the shortest path from u to v. Let $X = \{1, 2, 5\}$ the set of these bits. Any ordering of set X gets a different shortest path. For example $O_1 = (2, 5, 1)$ gets the path $< 01100 - 00100 - 00101 - 10101 >$ or $O_2 = (1, 5, 2)$ is equivalent to the path $< 01100 - 11100 - 11101 - 10101 >$.

As system of distinct representative (SDR) and Hall's theorem have been used in the proposed condition in [9], we review them here.

Definition 10. Let $(A_1, A_2, \ldots, A_m)$ be a collection of subsets of a set $A = \{a_1, a_2, \ldots, a_n\}$, $m \leq n$. An ordered set of distinct elements $[a_{i1}, a_{i2}, \ldots, a_{im}]$ is called a system of representatives (*SDR*) if $a_{ij} \in A_j$, for $1 \leq j \leq m$.

In other words, we are looking for one distinct element in each set as a representative. Clearly, it is not always possible to find a SDR for some sets. Hall's theorem, also known as Hall's marriage condition, gives necessary and sufficient condition for a collection of sets to have a SDR.

Theorem 2.1. *(Hall's Theorem) A collection of sets $A_1, A_2, \ldots, A_n$ has a SDR if and only if for every $k \geq 1$, and every set $\{i_1, i_2, \ldots, i_k\}$, $|\bigcup_{j=1}^{k} A_{i_j}| \geq k$.*

For instance, let $A = \{1, 2, 3, 4\}, A_1 = \{1, 2, 4\}, A_2 = \{1, 2\}, A_3 = \{2, 4\}, A_4 = \{2, 3, 4\}$. One possible SDR can be $[2, 1, 4, 3]$. If we change A_4 to $A_4 = \{1, 4\}$ there will be no SDR. The reason based on Hall's theorem is $|A_1 \cup A_2 \cup A_3 \cup A_4| = 3 \ngeq 4$.

Now the following theorem by [9] shows that for a collection of sets, the existence of SDR and disjoint ordering are equivalent.

Theorem 2.2. *For any collection of nonempty finite sets $X_1, X_2, \ldots, X_s$, in which all singletons are distinct, there is a disjoint ordering if and only if a SDR exists for the collection.*

Finally, the following theorem, gives the necessary and sufficient condition for node-disjoint shortest paths to exist in a hypercube.

Theorem 2.3. *Let v be a source node and $v_1, v_2, \ldots, v_s$ any other $s \leq n$ nodes on Q_n, necessarily distinct. For $1 \leq s \leq n$, let X_i denote the set of coordinate positions where v and v_i differ. A necessary and sufficient condition for there to be internally node-disjoint shortest paths from the source node v to $v_1, v_2, \ldots, v_s$ is when there exists a pairwise disjoint ordering for $X_1, X_2, \ldots, X_s$.*

The complete proof is presented in [9]. Briefly, as we reviewed, each ordering is a shortest path, and based on the definition, when two orderings are disjoint, it means they are node-disjoint paths. The authors also proposed an algorithm for constructing the path, i.e. pairwise disjoint ordering for $v_1, v_2, \ldots, v_s$ with $O(sn^3)$ time complexity. The algorithm is constructive based on induction. Briefly, it first reorders X_i's in ascending order by their size. Then it assigns to O_1 an arbitrary permutation of X_1. At each step, it constructs the ordering of X_k called O_k such that it meets all conditions mentioned in definition 9. The construction is either by adding one element to O_k or by reordering the previous orderings such that the current O_k is disjoint with any previous ordering. In this way, when we construct the final O_s we are sure all orderings are pairwise disjoint. The running time has since been improved in [2] to $O(n^3)$.

There is also some research on conditions for the existence of disjoint paths which are not necessarily shortest in the hypercube. For example, in [9] it is shown that there exist n disjoint paths from a given source node in an n-cube such that all paths have lengths at most no more than $n + 1$.

In the following sections, we discuss research around the problem of finding shortest path in the presence of blocking nodes.

2.4 Shortest Path Between Two Nodes in Faulty Hypercube

Since in practical applications hypercube networks usually consist of many processors, it is likely for the system to face failed component. Therefore, one of the most important paradigms in the hypercube is to design routing algorithms which tolerate faulty processors, also called faulty or blocking nodes.

In this thesis we only focus on one routing paradigm in faulty hypercubes, finding the shortest path between two given nodes if it exists. Let the Hamming distance of the two given nodes be r. Therefore, there are $r!$ different shortest paths where only some of them are using non-blocking nodes. Since checking all $r!$ is very time consuming we need to find more efficient routing algorithms.

This problem has been studied in two different perspectives based on the amount of information known to each individual node. In the first type, which can be considered as more theoretical, each node is aware of the current situation of all other nodes, i.e. each node knows the set of blocking nodes before sending its data. In the second

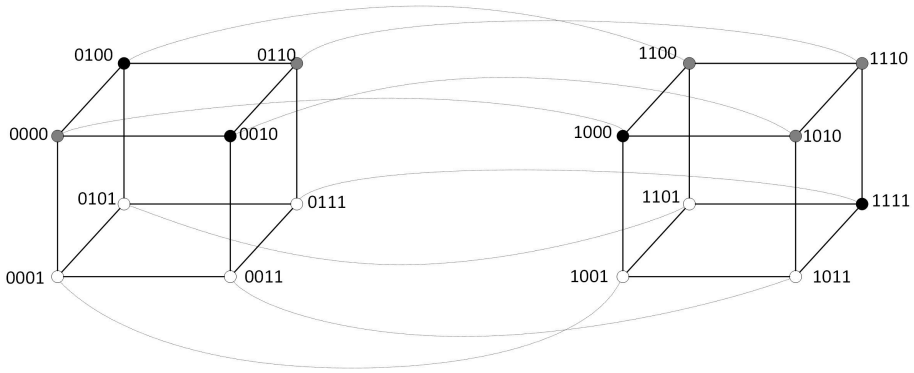

Figure 2.1: white: non-faulty nodes, black: faulty-nodes, grey: unsafe nodes

type, which is more natural, each node stores the information of its neighbours within a range. We first review some work done in the second type. We then discuss the first type studies, including our proposed algorithm.

2.5 Safe and Unsafe Nodes Method

A variety of routing algorithms have been proposed based on the idea of safe and unsafe nodes first introduced by Chiu and Wu [5]. Since these algorithms aim to solve the problem with limited amount of information in each node, the concept of safe and unsafe nodes helps to identify fault-free nodes that may cause data passing possible in a faulty hypercube. We review some of the proposed algorithms based on safe and unsafe nodes. In the following definition, fault-free nodes refer to nodes which are not originally in the set of faulty nodes.

Definition 11. A fault-free node is defined as an unsafe node if it has either two or more faulty nearest neighbours, or three or more faulty or unsafe nearest neighbours.[5]

A non-faulty node which is not unsafe is called a safe node. In [5] an algorithm is proposed to identify unsafe nodes in recursive manner. Briefly, at each iteration each node sends its current state to its neighbours and gets their states and updates its status as a safe or an unsafe node. As an example, a 4-cube with some faulty and unsafe nodes has been shown in Figure 2.1. The more unsafe nodes exist in the system, the faster the algorithm identifies all unsafe nodes.

In the following theorem, a *feasible path* refers to a path that traverses through no faulty nodes.

Theorem 2.4. *In a faulty hypercube, if node A is a safe node, for any non-faulty node*

B, there always exists a feasible minimum path of length equal to $H(A, B)$ between A and B. [5]

Kaneko and Ito [12] extended this approach and defined another notion of fully reachable node. Based on their definition a fully reachable node is a node which can reach all non-faulty nodes which have Hamming distance l from the node via paths of length l. There are many details and definitions in their approach which are beyond this thesis. Their approach is notable since it has a better performance compared to the one proposed by Chie and Wu [5].

2.6 Sufficient Conditions

There is some research that gives sufficient conditions for the shortest path between two nodes in a hypercube to exist in the presence of faulty processors, also referred to as blocking nodes. As mentioned in Chapter 1, since the hypercube is node and edge symmetric and has a recursive structure, we can safely assume that the two nodes are $s = 0^n$ and $t = 1^n$ as source and target nodes, respectively. For example, Assume two given nodes are $s = 01001$ and $t = 10011$ in a 5-cube. Since third and fifth bits are the same, we just need to change the first, second and forth bits. Therefore, we can solve the problem for a 3-cube from $s = 010$ to $t = 101$ and then by relabeling the nodes it will change to $s = 000$ and $t = 111$.

The following theorem is pointed out in [5] which can be applied when the number of blocking nodes is less than n.

Theorem 2.5. *In Q_n, $n \geq 2$, if the number of faulty nodes is less than n, then there exists a shortest path from s to t that does not intersect any of the blocking nodes.*

This theorem can be proven using induction on n. The next theorem proposed by Qiu, Noroozi, et al [3] is more generalized than the previous one and is for the case when the number of blocking nodes is less than $2(n-1)$ with two obvious conditions.

Theorem 2.6. *In Q_n, if the number of blocking nodes (that are represented by b_i's) is less than $2(n-1)$, there exists a shortest path from $s = 0^n$ to $t = 1^n$ that does not intersect any of the blocking nodes if for any subset of size n of the blocking nodes, these two conditions hold*
(a) $\{b_1, b_2, \cdots, b_n\} \neq \{011...11, 101...11, \cdots, 111...10\}$ and
(b) $\{b_1, b_2, \cdots, b_n\} \neq \{00...001, 00...010, \cdots, 10...000\}$,
i.e., at least one neighbour of s is not a blocking node and at least one neighbour of t is not a blocking node.

Note that clearly the shortest path can exist even when the number of blocking nodes is greater than $2(n-1)$ and Theorem 2.6 is only a sufficient condition. In the next section we review another routing algorithm in the presence of faulty nodes in the hypercube.

2.7 Indentification Algorithm

Qiu, Noroozi, et al [3] studied the problem of shortest path between two nodes in a faulty hypercube and proposed a method called *identification* algorithm. The proposed algorithm is based on finding non-blocking nodes that are not reachable via source node. In other words, there are some nodes which are not in the given set of blocking nodes, but they are also not reachable from source node, called *dead end node* in the paper. At the end of the algorithm if the target node, source node and all neighbours of the source node are not dead end nodes, it implies that a shortest path exists from source to target. For convenience, here 1^n is assumed as the source node and 0^n as the target node. More formally:

Theorem 2.7. *In Q_n with m blocking nodes, $1 \leq m \leq 2^n - (n+1)$, there exists a shortest path from $s = 1^n$ to $t = 0^n$ that does not intersect any of the blocking nodes if and only if after performing the identification algorithm, at least one of the 0^n neighbours is a non-blocking node (neither original blocking node nor dead end node). [3]*

The *identification* algorithm is based on the fact that some blocking nodes with the same Hamming weight, say h, (i.e. number of bits that are 1 in the binary representation) can block all the paths of some nodes with Hamming weight $h-1$. This makes some new blocking nodes or dead end nodes. More precisely, it has been proven that if bit-wise AND operation of two blocking nodes (including already identified dead end nodes) with the Hamming weight h results in a node with the Hamming weight $h-1$, the new node is a dead end node and will be added to the set of blocking nodes. For instance, in Q_5 let $b_1 = 10110$ and $b_2 = 11010$ be two blocking nodes whose Hamming weight is 3. Since $v = (b_1 \wedge b_2) = 10010$ has Hamming weight 2 it will be unreachable from the target node.

The algorithm starts by performing bit-wise AND operation on every pair of blocking nodes with Hamming weight $n-1$. There are possibly $\binom{h(n-1)}{2}$ pairs, where $h(n-1)$ is the number of blocking nodes with Hamming weight of $n-1$. If the result of each AND operation is a node with Hamming weight of $n-2$ it will be added

to the set of blocking nodes. Next time, it will take every combination of all three nodes of Hamming weight n-2 and does the same operation on them, then on every combination of all four nodes of Hamming weight n-3, etc. At the end, if there is at least one neighbour of 0^n that is not in the set of blocking nodes it implies that there is a shortest path from 1^n to 0^n.

In the worst case the *identification* algorithm can have an exponential running time in terms of the number of the blocking nodes, m. Since at each step the number of operations is $\binom{h(n-k)}{k+1}$ with $n-k$ the Hamming distance of blocking nodes at step k. Therefore in worst case the total number of operations is $\sum_{k=1}^{n-2}\binom{h(n-k)}{k+1}$ where $h(n-k)$ can possibly be $\binom{n}{n-k}$.

Chapter 3

Proposed Algorithm

3.1 Introduction

In this chapter we design an algorithm for finding a shortest path between two nodes in a hypercube when some nodes have become faulty and thus cannot be used. As mentioned in previous chapter, due to symmetry and recursive structure in hypercube, we can safely assume that the source node is 0^n and the target node is 1^n in an $n-$dimensional hypercube or $n-$cube. More formally, the problem can be stated as follows:

Given source and target nodes $s = 0^n$ and $t = 1^n$ in an n-dimensional hypercube, and m blocking nodes b_1, b_2, ..., b_m, $b_i \neq s$, $b_i \neq t$, $1 \leq i \leq m$, does there exist a shortest path from s to t that does not intersect any blocking node?

We first explain how to model the problem to sets and permutations using a similar technique used in [9]. Then we explain a naive exponential approach to solve the modeled problem and finally we present and discuss our proposed polynomial algorithm.

3.2 Ordering Model of Shortest Path in Presence of Faulty Nodes

As discussed in Chapter 1, in an n-dimensional hypercube, nodes are n-tuples of 0's and 1's of length n. Therefore, a node in n-cube can be represented as $v = v_1 v_2 \dots v_n, v_i \in \{0, 1\}$. In our modeling, we represent each node as a set as follows:

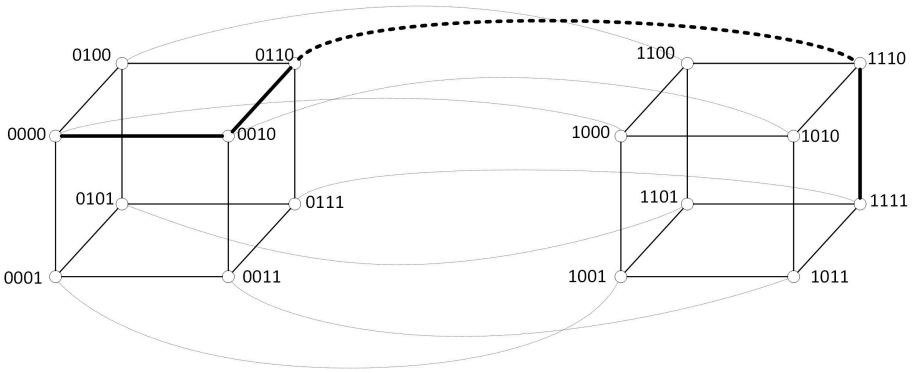

Figure 3.1: The path of Example 3.2 in a 4-cube

Definition 12. A node $v = v_1v_2 \dots v_n, v_i \in \{0, 1\}$ in an n-cube is represented by $X = \{i | v_i = 1, 1 \leq i \leq n\}$. In other words, X is the set of all *positions* (bits) in v where their values are 1.

Example 3.1. In a 5-hypercube, the set representations of some example nodes are as follows:
$v_1 = 01001 \quad X_1 = \{2, 5\}$
$v_2 = 10101 \quad X_2 = \{1, 3, 5\}$
$v_3 = 11100 \quad X_3 = \{1, 2, 3\}$

A path can also be represented using a permutation of bit positions. Before giving the formal definition of a path using permutation, we need some notations as follows:

Definition 13. For a permutation $P = [p_1, p_2, \dots, p_n]$, we define partial permutation $P_k = [p_1, p_2, \dots, p_k]$ and set of partial permutations as $\{P_k\} = \{p_1, p_2, \dots, p_k\}$.

For instance if $P = [3, 1, 4, 2, 5]$ then $P_3 = [3, 1, 4]$ and $\{P_3\} = \{1, 3, 4\}$ (elements are given in a sorted order here).

Two nodes are adjacent in a hypercube if and only if their tuple representations (or binary representations) differ only in one position. This makes it possible to show a simple path from 0^n to 1^n in a hypercube with a permutation and the set representation of nodes.

Definition 14. A path in an n-cube in the form of $< 0^n, v_1, v_2, \dots, v_{n-2}, 1^n >$ can be shown as permutation $P = [p_1, p_2, \dots, p_n]$ such that $\{P_k\} = X_k$ where X_k is the set representation of v_k. Note that since the path should be a simple path, therefore for $i \neq j, v_i \neq v_j$. Also we will use $\{P\}$ and $|P|$ notations as set of elements of permutation P and its size, respectively.

Example 3.2. A paths in a 4-dimensional hypercube shown in Figure 3.1 can be represented with a permutation as follows:
$< 0000, 0010, 0110, 1110, 1111 > \longleftrightarrow P = [3, 2, 1, 4]$

Using these notations, the following theorem models the problem to a permutation problem:

Theorem 3.1. *In an n-dimensional hypercube with m blocking nodes $X_1, X_2, \ldots, X_m$ in which X_i is the set representation of a blocking node, there exists a shortest path from 0^n to 1^n if and only if there is a permutation P of numbers $\{1, 2, \ldots, n\}$ such that:*

$$\forall i : 1 \leq i \leq m : X_i \neq \{P_{|X_i|}\} \textit{ (}\textbf{\textit{validation condition}}\textit{)}$$

Proof. Any permutation of the set $\{1, 2, \ldots, n\}$ is a simple path from 0^n to 1^n. Also it is a shortest path because its length is n. Since each $\{P_k\}$ is a node of the path, therefore if validation conditions hold for P, it implies that none of the nodes of the path are in blocking node set. □

Example 3.3. In a 4-cube, $m = 4$
$X_1 = \{1, 3, 4\}$
$X_2 = \{2, 4\}$
$X_3 = \{1\}$
$X_4 = \{3, 4\}$
One possible answer as depicted in Figure 3.2 could be $P = [2, 3, 1, 4]$ since:
$\{P_3\} = \{1, 2, 3\} \neq X_1$
$\{P_2\} = \{2, 3\} \neq X_2, X_4$
$\{P_1\} = \{2\} \neq X_3$

Note that for the rest of the thesis, in all figures, paths and blocking nodes are depicted with black lines and black nodes, respectively. Before explaining our proposed polynomial algorithm for finding the permutation P, we first review a naive inefficient approach.

3.2.1 Naive Approach

A simple solution to this model would be creating all possible $n!$ permutations of P and then checking each of them against all X_i's to see if it satisfies the validation condition or not. Therefore, the time complexity would be $O(n! \times m)$.

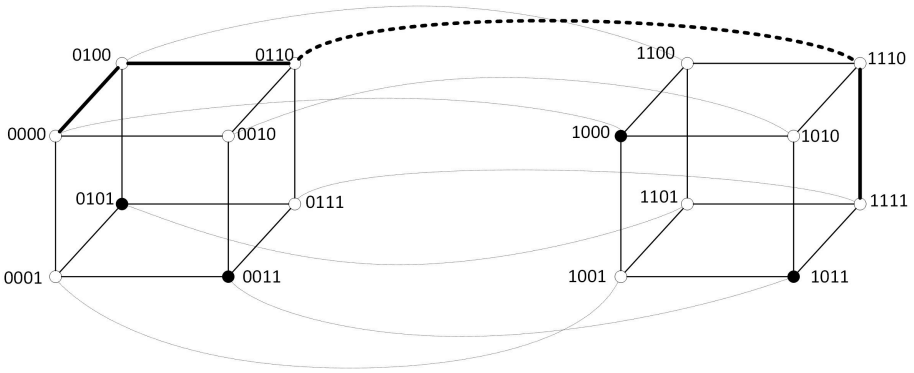

Figure 3.2: 4-cube of Example 3.3

3.3 The Proposed Polynomial Algorithm

3.3.1 Big Picture

Our proposed algorithm is based on creating the elements of P one by one. In other words, we start with an empty permutation $P = [\ ]$ and we choose the first element, say α, from $N = \{1, 2, 3, .., n\}$. If among all possible $(n-1)!$ permutations that start with $[\alpha]$, there exists at least one permutation that satisfies the validation condition, we fix α as the first element of the path. We repeat the same process for the second element, say β, among remaining numbers from N. In this case, we need to make sure that there is a permutation that starts with $[\alpha, \beta]$ and satisfies the validation conditions among all possible $(n-2)!$ permutations. We follow the same process for all positions in P until we create the final permutation. Clearly, if we cannot find α in the first step, it means there is no such permutation and a shortest path does not exist.

The heart of the algorithm is how to efficiently make sure that the current partial permutation (for example $[\alpha, \beta]$) can lead to an answer. For the rest of the chapter, we refer to *valid permutation* as a permutation which satisfies the validation condition.

To make sure that the current partial permutation P leads to a valid permutation we use the inclusion-exclusion principle. We first count all invalid permutations that start with P and then subtract them from all possible permutations starting with P to get the valid permutations (complement law). If the number of valid permutations is more than zero, it means the current partial permutation leads to a valid permutation and we can safely continue extending the partial permutation.

In Algorithm 1, NumberOfValidPermutations(X,P,n) is a function that computes the number of total valid permutations that can be made with extending P as the partial permutation considering X as blocking nodes. Naively, this function can

be computed using a simple inclusion-exclusion counting. Since applying inclusion-exclusion principle on a set of size m has time complexity of $O(2^m)$ we need to optimize the idea. Optimization is based on creating a directed acyclic graph (DAG) out of blocking nodes and leveraging dynamic programming (DP) on the DAG to efficiently count inclusion-exclusion terms. Also in Algorithm 1, appending an element, say α to permutation P means adding α to P as the last element.

We discuss each step with details in the following sections.

Algorithm 1 Big Picture of the Proposed Algorithm

```
1: procedure FindingShortestPath(X, n)          ▷ X: set of blocking nodes, n:dimension of cube
2:     P ← [ ]                                  ▷ Start with an empty ordering
3:     for i = 1 → |N| do
4:         α ← N_i                              ▷ Get i-th element of N
5:         P ← P + [α]                          ▷ Append α to the end of P
6:         if NumberOfValidPermutations(X, P, n) > 0 then
7:             N ← N \ {α}                      ▷ Remove α from the set N
8:             i ← 1                            ▷ Reset to the beginning of N
9:         else
10:            P ← P \ [α]                      ▷ Remove α from the end of P
11:        end if
12:        if α == n and P == [] then           ▷ Reached n but no first element yet
13:            break                            ▷ There is no answer
14:        end if
15:        if α == n and P == [] then           ▷ Reached n but no first element yet
16:            break                            ▷ There is no answer
17:        end if
18:    end for
19:    return P                                 ▷ In case of no answer, P will be empty
20: end procedure
```

3.3.2 Counting Valid Permuations

Assume we have added a new element to our partial permutation P. Now we have to make sure that the current permutation leads to a valid permutation. As mentioned, we use the inclusion-exclusion principle to count all invalid permutations that start with the current permutation. Then we subtract the number of all invalid permutations from all possible permutations to get the number of valid ones. Finally, if the number of valid permutations is more than zero, it means the current partial permutation can lead to a valid permutation.

In general, the inclusion-exclusion principle for counting the number of elements in the union of m sets is as follows:

$$\begin{aligned}\left|\bigcup_{i=1}^{m} Y_i\right| = &\sum_{i_1=1}^{m} |Y_{i_1}| \\ &- \sum_{1\le i_1<i_2\le m} |Y_{i_1} \cap Y_{i_2}| \\ &+ \sum_{1\le i_1<i_2<i_3\le m} |Y_{i_1} \cap Y_{i_2} \cap Y_{i_3}| \\ &- \cdots (-1)^{p-1} \sum_{1\le i_1<\cdots<i_p\le m} \left|Y_{i_1} \cap \cdots \cap Y_{i_p}\right| \cdots \\ &(-1)^{m-1} |Y_1 \cap Y_2 \cap \cdots \cap Y_m|.\end{aligned} \tag{3.1}$$

In our problem, we define Y_i as follows:

Definition 15. Y_i = set of all permutations that start with the current partial permutation P, and violate the *validation condition* for X_i.

Based on Definition 15, each Y_i violates the *validation condition* for at least X_i. Therefore $\bigcup_{i=1}^{m} Y_i$ has all invalid permutations and thus $\left|\bigcup_{i=1}^{m} Y_i\right|$ gives the number of all invalid permutations starting with P. The total number of permutations that start with P is $(n - |P|)!$. Thus we have:

$$\textit{number of valid permutations} = (n - |P|)! - \left|\bigcup_{i=1}^{m} Y_i\right| \tag{3.2}$$

Example 3.4. Based on Definition 15, for a 4-cube with 4 blocking nodes and $P = [4]$, Y_i's are as follows:
$n = 4, m = 4, P = [4]$
$X_1 = \{1, 3, 4\}$ $\quad Y_1 = \{[4, 1, 3, 2], [4, 3, 1, 2]\}$
$X_2 = \{2, 4\}$ $\quad Y_2 = \{[4, 2, 1, 3], [4, 2, 3, 1]\}$
$X_3 = \{1\}$ $\quad Y_3 = \{\}$
$X_4 = \{3, 4\}$ $\quad Y_4 = \{[4, 3, 1, 2], [4, 3, 2, 1]\}$

In the above example $\bigcup_{i=1}^{4} Y_i$ has 5 elements. On the other hand, the number of all possible permutations starting with $P = [4]$ is equal to $(4-1)! = 6$. Therefore, the total number of valid permutations is $6 - 5 = 1$ which means there is a permutation starting with [4] and that leads to an answer, namely $P = [4, 1, 2, 3]$ as illustrated in Figure 3.3.

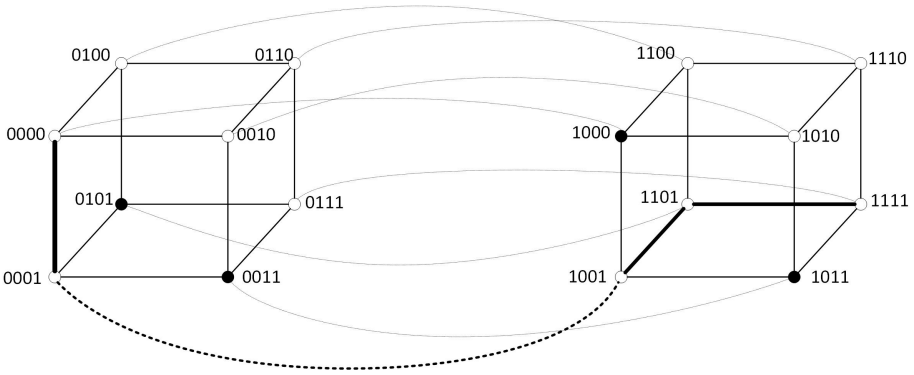

Figure 3.3: A solution for the Example 3.4

Since in our inclusion-exclusion formula we only need the size of the sets, we do not actually need to compute all of their members. This can be done directly with the following theorem:

Theorem 3.2. *Calculating the size of Y_i can be directly done using following formula:*

$$|Y_i| = \begin{cases} (|X_i| - |P|)! * (n - |X_i|)! & (If\ \{P\} \subseteq X_i) \\ 0 & (If\ \{P\} \not\subseteq X_i) \end{cases} \tag{3.3}$$

Proof. If $\{P\} \not\subseteq X_i$, there is no permutation that starts with P's elements and X_i's elements at the same time since there is at least one element in P that is not in Y_i and thus violates the definition of Y_i that all elements of set Y_i must start with P's elements. Also the ordering of common elements of P and Y_i is fixed and equal to P. Then, any permutation of all numbers of X_i which have not been used by P can come next in $(|X_i|-|P|)!$ different ways. And finally any permutation of all remaining numbers from N which have not been used yet can come next in $(n - |X_i|)!$ different ways.

□

As an example, consider the following blocking nodes and partial permutation:

Example 3.5. In a 6-cube:
$n = 6, P = [1, 3]$
$X_1 = \{1, 3, 4\}$
$Y_1 = \{[1, 3, 4, 2, 5, 6], [1, 3, 4, 5, 2, 6], [1, 3, 4, 6, 5, 2],$
$[1, 3, 4, 5, 6, 2], [1, 3, 4, 2, 6, 5], [1, 3, 4, 6, 2, 5]\}$

$|Y_1| = (|X_1| - |P|)! * (n - |X_1|)! = (3-2)! * (6-3)! = 1 * 3! = 6$

$X_2 = \{1, 3, 5, 2\}$
$Y_2 = \{[1, 3, 5, 2, 4, 6], [1, 3, 5, 2, 6, 4],$
$[1, 3, 2, 5, 4, 6], [1, 3, 2, 5, 6, 4]\}$
$|Y_2| = 4 = (|X_2| - |p|)! * (n - |X_2|)! = 2! * 2! = 4$

In the inclusion-exclusion formula we also need to count the size of intersection of multiple Y_i's. This is also a generalization of finding the size of each Y_i individually. Consider X_i and X_j and without loss of generality assume $|X_i| \leq |X_j|$. The same conditions that we have for single form, apply here as well.

Lemma 3.1. *Assuming* $|X_i| \leq |X_j|$*:* $|Y_i \cap Y_j| > 0$ *iff* $\{P\} \subseteq X_i \subseteq X_j$

In other words, Lemma 3.1 says for two Y_i and Y_j to have non-empty intersection, one needs to be a subset of the other one. It can be proved with the same logic as Theorem 3.2. We generalize Lemma 3.1 for k sets and prove it.

Theorem 3.3. *Assuming* $|X_1| \leq |X_2| \leq \cdots \leq |X_k|$*, the size of the intersection of corresponding* Y_i*'s,* $|Y_1 \cap Y_2 \cap \cdots \cap Y_k|$*, can be calculated as follows:*

$$|\bigcap_{i=1}^{k} Y_i| = \begin{cases} (|X_1| - |P|)! * (|X_2| - |X_1|)! * \dots \\ *(|X_k| - |X_{k-1}|)! * (n - |X_k|)! & \text{if } \{P\} \subseteq X_1 \subseteq X_2 \subseteq \cdots \subseteq X_k \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

Proof. We use induction over k to prove it. The base case is $k = 1$ which is equal to Theorem 3.2 that has been already proved. For the induction step, assume we have the answer of $\{P\} \subseteq X_1 \subseteq X_2 \subseteq \dots X_{k-1}$ and call it P'. Now all we have is P' and X_k which have the same conditions as Theorem 3.2 that has already been proved. □

As an example, consider the intersection of two blocking nodes as follows:

Example 3.6. In a 6-cube:
$n = 6, m = 2, P = [1]$
$X_1 = \{1, 3, 4\}$
$X_2 = \{1, 3, 4, 6\}$

$$
\begin{aligned}
Y_1 = \{&[1,4,3,2,5,6],[1,4,3,2,6,5],[1,4,3,6,2,5],\\
&[1,4,3,6,5,2],[1,4,3,5,2,6],[1,4,3,5,6,2],\\
&[1,3,4,2,5,6],[1,3,4,2,6,5],[1,3,4,6,2,5],\\
&[1,3,4,6,5,2],[1,3,4,5,2,6],[1,3,4,5,6,2]\}\\
Y_2 = \{&[1,3,4,6,2,5],[1,3,6,4,2,5],[1,6,3,4,2,5]\\
&[1,6,4,3,2,5],[1,4,3,6,2,5],[1,4,6,3,2,5]\\
&[1,3,4,6,5,2],[1,3,6,4,5,2],[1,6,3,4,5,2]\\
&[1,6,4,3,5,2],[1,4,3,6,5,2],[1,4,6,3,5,2]\}
\end{aligned}
$$

$$\{Y_1 \cap Y_2\} = \{[1,3,4,6,2,5],[1,3,4,6,5,2],[1,4,3,6,2,5],[1,4,3,6,5,2]\}$$
$$|Y_1 \cap Y_2| = (|X_1| - |p|)! * (|X_2| - |X_1|)! * (n - |X_2|)! = 2! * 1! * 2! = 4$$

Based on Theorem 3.3, each term in the inclusion-exclusion formula can be calculated directly without constructing the set itself. Although many terms are zero, but still the number of non-zero terms are in order of $O(2^m)$ which leads to an exponential solution if we do counting without any other optimization. That is why we continue optimizing the solution by creating a DAG out of blocking nodes and leveraging dynamic programming on it.

3.3.3 Creating DAG of sets

Based on Theorem 3.3, the only non-zero terms are those whose members make a chain of subset relation. On the other hand, we know that subset relation is an acyclic relation for proper subsets, therefore we can create a directed acyclic graph based on X_i's.

Algorithm 2 Creating Graph Out of Blocking Nodes

```
1:  procedure CreateGraph(X)
2:      m ← |X|
3:      G ← empty graph with X as the set of vertices
4:      for i = 1 → m do
5:          for j = 1 → m do
6:              if i ≠ jand X_i ⊂ X_j then
7:                  add edge (X_i, X_j) to E(G)
8:              end if
9:          end for
10:     end for
11:     return G
12: end procedure
```

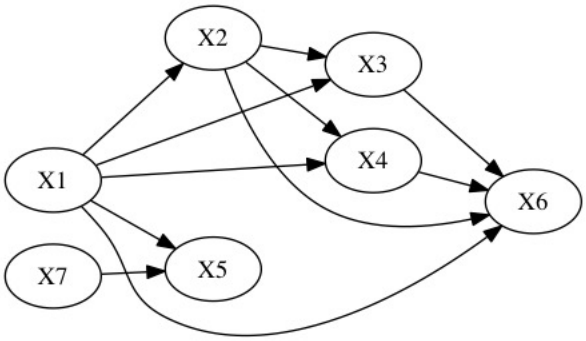

Figure 3.4: DAG of Example 3.7

Definition 16. Graph $G(X, E)$ is a directed graph with set of blocking nodes X as vertices and there is a directed edge from X_i to X_j iff $X_i \subset X_j$.

In this graph we are interested in paths since they connect those blocking nodes that make a chain of subset relation. Therefore to connect Theorem 3.3 and paths in the graph we define *value of path* as follows:

Definition 17. For a path in the form of $< X_1, X_2, \ldots, X_k >$ we define the *value of the path* to be $V[< X_1, X_2, \ldots, X_k >]=|Y_1 \cap Y_2 \cap \cdots \cap Y_k|$.

In other words, the *value of a path* is equal to number of permutations that violate the *validation condition* for all (blocking) nodes involved in the path.

Example 3.7. In a 6-cube:
$n = 6, m = 7, P = [2]$
$X_1 = \{1, 2\}$
$X_2 = \{1, 2, 4\}$
$X_3 = \{1, 2, 4, 5\}$
$X_4 = \{1, 2, 3, 4\}$
$X_5 = \{1, 2, 3, 6\}$
$X_6 = \{1, 2, 3, 4, 5\}$
$X_7 = \{6\}$
$V[< X_1, X_4, X_6 >] = |Y_1 \cap Y_4 \cap Y_6| = 1! * 2! * 1! * 1! = 2$

As a result of Definition 17, each term of the inclusion-exclusion formula can be calculated by summing values of corresponding paths. More formally:

Theorem 3.4. *Each term of the inclusion-exclusion formula can be calculated by the value of the paths.*

$$\sum_{1\leq i_1<\cdots<i_k\leq n} |Y_{i_1}\cap\cdots\cap Y_{i_k}| = \sum_{1\leq i_1<\cdots<i_k\leq n} V[<X_{i_1},\ldots,X_{i_k}>] \text{ if such path exists} \tag{3.5}$$

Therefore, for calculating the inclusion-exclusion formula all we need is to calculate sum of values of all paths with size k for $1 \leq k \leq n$.

Example 3.8. Based on Figure 3.4:

$\sum_{1\leq i_1<i_2<i_3\leq 6} |Y_{i_1}\cap Y_{i_2}\cap Y_{i_3}| =$
$V[<X_1,X_2,X_3>]+V[<X_1,X_2,X_4>]+V[<X_1,X_2,X_6>]+V[<X_1,X_4,X_6>]$
$+V[<X_2,X_3,X_6>]+V[<X_2,X_4,X_6>]$

If we calculate the value of all paths one by one and then sum them up, we still need an exponential number of calculations. There are three observations that make it possible to use dynamic programming to sum V values efficiently. Firstly, we are working on a DAG which means there is no cycle and all paths are simple paths. Secondly, a path in a graph has inherently a recursive definition; i.e. each path of size k is created by a path of size $k-1$. Thirdly, we will show that *the value of a path* can also be calculated recursively.

3.3.4 Dynamic Programming on DAG

Example 3.9. Based on Theorem 3.3, V values of Figure 3.4 can be calculated as follows:

$$\begin{aligned}
V[<X_1>] &= (|X_1|-|P|)!*(n-|X_1|)\\
V[<X_1,X_2>] &= (|X_1|-|P|)!*(|X_2|-|X_1|)!*(n-|X_2|)!\\
&= \frac{V[<X_1>]}{(n-|X_1|)!}*(|X_2|-|X_1|)!*(n-|X_2|)!\\
V[<X_1,X_2,X_6>] &= (|X_1|-|P|)!*(|X_2|-|X_1|)!*(|X_6|-|X_2|)!*(n-|X_6|)!\\
&= \frac{V[<X_1,X_2>]}{(n-|X_2|)!}*(|X_6|-|X_2|)!*(n-|X_6|)!
\end{aligned}$$

As shown in Example 3.9, the value of a path of size k, $V[<X_1,X_2,\ldots,X_k>]$, can be calculated based on the value of a path of size $k-1$, $V[<X_1,X_2,\ldots,X_{k-1}>]$.

To make the formula more clear, we define a new notation, V' as follows:

Definition 18. $V'[< X_1, X_2, \ldots, X_k >] = \frac{V[<X_1,X_2,\ldots,X_k>]}{(n-|X_k|)!}$

In other words, $V[< X_1, \ldots, X_k >]$ is defined on all members of set N, while $V'[< X_1, \ldots, X_k >]$ has the same definition as V but on the members of X_k. Therefore, in V' there is no permutation of numbers that are in N but not in X_k, hence in the definition of V' we divided V by $(n - |X_k|)!$ to remove those permutations.
Based on Definition 17, Definition 18 and Thoerem 3.3 the following theorem gives the recursive definition for V':

Theorem 3.5. *V' can be calculated recursively as follows:*

$$V'[< X_1, \ldots, X_k >] = \begin{cases} 0 & k = 1, \{P\} \not\subseteq X_k \\ (|X_k| - |P|)! & k = 1, \{P\} \subseteq X_k \\ V'[< X_1, \ldots, X_{k-1} >] * (|X_k| - |X_{k-1}|)! & k > 1 \end{cases} \tag{3.6}$$

The recurrence relation of Theorem 3.5 is based on Theorem 3.3 and basically with the definition of V' it is very similar to cumulative product. To sum up all V' values of paths with size k, we use dynamic programming by defining $T(X_i, k)$ as follows:

Definition 19. $T(X_i, k)$ = summation of V' value of all paths of size k that end at node X_i. More formally:

$$T(X_i, k) = \sum_{1 \le s_1 < \cdots < s_{k-1} \le m} V'[< X_{s_1}, X_{s_2}, \ldots, X_{s_{k-1}}, X_i >] \text{ if such a path exists} \tag{3.7}$$

The approach of dynamic programming is based on calculating the answer node by node. Using result of Theorem 3.5, calculating V' value of a path of size k can be done using the answer of the paths of size $k - 1$. Therefore for calculating $T(X_i, k)$ all we need is $T(X_j, k-1)$ such that there is an edge from X_j to X_i. The recurrence relation is as follows:

$$T(X_i, k) = \begin{cases} 0 & k = 1, \{P\} \not\subseteq X_i \\ (|X_i| - |P|)! & k = 1, \{P\} \subseteq X_i \\ \sum_{(X_j, X_i) \in E(G)} (T(X_j, k-1) * (|X_i| - |X_j|)!) & k > 1 \end{cases} \tag{3.8}$$

Implementation of $T(X_i, k)$ can be done efficiently similar to the known Bellman-Ford algorithm [7]. The idea is for calculating $T(X_i, k)$ order of nodes does not matter and we only need to make sure that paths of size $k-1$ have already been computed. We first iterate over k and then for each edge we update the value of its corresponding vertices. Therefore, with two nested loops all elements of T table will be computed efficiently. Also note that since we are dealing with a DAG, we do not need to handle cycles.

Algorithm 3 Computing Dynamic Programming Table

```
1: procedure ComputeDP(G, n, P)                ▷ G:Graph,P:Partial Permutation
2:     m ← |V(G)|
3:     for i = 1 → m do
4:         if {P} ⊆ X_i then
5:             T(X_i, 1) ← (|X_i| − |P|)!
6:         else
7:             T(X_i, 1) ← 0
8:         end if
9:     end for
10:    for k = 2 → n do
11:        for each edge (X_j, X_i) ∈ E(G) do
12:            T(X_i, k) ← T(X_i, k) + (T(X_j, k − 1) ∗ (|X_i| − |X_j|)!)
13:        end for
14:    end for
15:    return T
16: end procedure
```

3.3.5 Efficient Inclusion-exclusion Counting

The main problem with directly calculating the inclusion-exclusion formula is that a term in the form of $\sum_{1\leq i_1<\cdots<i_k\leq m} |Y_{i_1} \cap \cdots \cap Y_{i_k}|$ has $\binom{m}{k}$ sub-terms and since we need to calculate such terms for all $1 \leq k \leq n$, we would need to calculate $\sum_{i=1}^{m} \binom{m}{i} = 2^m - 1$ different terms which leads to an exponential time complexity.

We review Theorem 3.3, Definition 18 and Definition 19 to see how $T(X_i, k)$ is related to inclusion-exclusion terms:

Based on Theorem 3.3:

$$\sum_{1\leq i_1<\cdots<i_k\leq n} |Y_{i_1} \cap \cdots \cap Y_{i_k}| = \sum_{1\leq i_1<\cdots<i_k\leq n} V[< X_{i_1}, \ldots, X_{i_k} >] \text{ if such path exists}$$

And Definition 18:

$$V'[< X_1, X_2, \ldots, X_k >] = \frac{V[< X_1, X_2, ..., X_k >]}{(n - |X_k|)!}$$

Therefore:

$$\sum_{1 \le i_1 < \cdots < i_k \le n} |Y_{i_1} \cap \cdots \cap Y_{i_k}| = \sum_{1 \le i_1 < \cdots < i_k \le n} (V'[< X_{i_1}, \ldots, X_{i_k} >] * (n - |X_{i_k}|)!) \quad (3.9)$$

And based on Definition 19:

$$T(X_i, k) = \sum_{1 \le s_1 < \cdots < s_{k-1} \le m} V'[< X_{s_1}, X_{s_2}, \ldots, X_{s_{k-1}}, X_i >] \text{ if such a paths exists}$$

Thus finally we have:

$$\sum_{1 \le i_1 < \cdots < i_k \le m} |Y_{i_1} \cap \cdots \cap Y_{i_k}| = \sum_{i=1}^{m} (T(X_i, k) * (n - |X_i|)!) \quad (3.10)$$

Since the dynamic programming table can be calculated in polynomial time, therefore using Equation 3.10 we can reduce exponential time of calculating inclusion-exclusion formula to polynomial time.

Algorithm 4 Inclusion-Exclusion Counting

```
1:  procedure InclusionExclusionCounting(G, n, P)
2:      m = |V(G)|
3:      if {P} ∈ V(G) then          ▷ If {P} is equal to one of the nodes, i.e. X_i's
4:          return 0                ▷ There would be no answer
5:      end if
6:      T ← computeDP(G, P)
7:      total ← (n − |P|)!          ▷ All permutations starting with P
8:      sign ← −1                   ▷ For handling the sign of terms in ex.-in. formula
9:      for i = 1 → n do
10:         for j = 1 → m do
11:             total ← total + sign * T(X_j, i) * (n − |X_j|)!
12:         end for
13:         sign ← sign * −1
14:     end for
15:     return total
16: end procedure
```

3.3.6 Proposed Algorithm

As explained in Section 3.3.1, in the proposed algorithm we find the order of the elements of P one by one, i.e. we first make sure about the first element, and then we continue to the second element and so on. To make sure that each element leads to a valid permutation, we use the inclusion-exclusion formula.

In Algorithm 5 Line 14 checks whether after visiting all elements, P is still empty or not. If it is, that means no first element has been found which implies there is no answer.

Also, we create the graph of blocking nodes before going into the loop of finding elements since the definition of graph is independent from P.

Algorithm 5 Proposed Algorithm: Finding shortest path in the presence of blocking nodes

```
1:  procedure FindingShortestPath(X, n)          ▷ X: set of blocking nodes, n:dimension of cube
2:      P ← []                                   ▷ Start with an empty partial permutation
3:      N ← {1, 2, ..., n}
4:      G ← CreateGraph(X)
5:      for i = 1 → |N| do
6:          α ← N_i                              ▷ Get i-th element of N
7:          P ← P + [α]                          ▷ Append α to the end of P
8:          if InclusionExclusionCounting(G, n, P) > 0 then
9:              N ← N \ {α}                      ▷ Remove α from set N
10:             i ← 1                            ▷ Reset to beginning of N
11:         else
12:             P ← P \ [α]                      ▷ Remove α from the end of P
13:         end if
14:         if α == n and P == [] then           ▷ Reached n but no first element yet
15:             break                            ▷ There is no answer
16:         end if
17:     end for
18:     return P                                 ▷ In case of no answer, P will be empty
19: end procedure
```

3.3.7 Performance

As mentioned, we use m for the number of blocking nodes, and n for the dimension of the hypercube. Therefore, the performance of each section of the proposed algorithm is as follows:

- **Creating the Graph**. Algorithm 2 has two nested loops each over m. In the inner loop, we need to check whether the set X_i is subset of the set X_j. To do so efficiently, we can store all elements of X_j in a hash table and then iterate over X_i's elements and find each of them in the hash table. If there is at least one element of X_i which is not in the hash table then X_i is not a subset of X_j, otherwise it is. Since looking up a hash table takes $O(1)$ time, thus the entire subset checking can be done linearly $O(n)$. Finally, creating the graph can be done in $O(nm^2)$.

- **Dynamic Programming**. In Algorithm 3 steps 3-9 can be done in $O(mn)$ time with the same approach we described in creating the graph for checking $\{P\} \subseteq X_i$. The dominant cost is at Steps 10-15 which have two nested loops; one over n and one over the number of edges in graph. Since the graph has m nodes, therefore the maximum number of edges is in the order of $O(m^2)$. As a result, the total cost for Algorithm 3 is $O(nm + nm^2) = O(nm^2)$.

- **Inclusion-exclusion Counting**. In Algorithm 4 Step 3 can be done in time $O(nm)$ by checking every member of P in all m blocking nodes. Step 6 calls Algorithm 3 that has $O(nm^2)$ time complexity and is the dominant step of inclusion-exclusion counting. Steps 9-14 have only two nested loops with an $O(1)$ operation inside that needs $O(nm)$ operations overall. Therefore, the total cost of this algorithm is $O(nm + nm^2 + nm) = O(nm^2)$.

- **Finding the Shortest Path**. Algorithm 5 which is the main body of our proposed algorithm, can be simply analyzed as follows. Step 4, creating the graph, has time $O(nm^2)$. Steps 5-17 have one loop with at most n iterations which calls inclusion-exclusion each time. Thus, the loop performs in time $O(n \times (nm^2)) = O(n^2m^2)$. Therefore, the proposed algorithm can be done in time of $(nm^2 + n^2m^2) = O(n^2m^2)$ which is polynomial in terms of number of faulty nodes and dimension of the hypercube.

3.3.8 Performance Optimization

In the proposed algorithm section, we create the graph once and compute the dynamic programming table on it in every iteration. Based on Theorem 3.3, if the set of partial permutations $\{P\}$ is not a subset of a node X_i, $T(X_i, k)$ for all $1 \leq k \leq n$ will be zero. Thus, in two nested iterations of the dynamic programming algorithm, many terms are calculated as zero. Therefore, we can safely remove all nodes X_i where

$\{P\} \not\subseteq X_i$ and reduce the size of the graph in order to calculate table T faster. Note that the effectiveness of this optimization is based on the blocking nodes and thus it cannot change the time complexity. An implementation of the optimized version of the proposed algorithm is shown in Algorithm 6. In Step 8, we keep a copy of the graph before removing nodes based on the current P. If that extension of P leads to a valid permutation, we continue with the updated graph (since we will not need those removed nodes anymore), otherwise we replace the graph with the unchanged one. As we go forward finding P's elements, the size of the graph will reduce accumulatively. Therefore, in many cases this optimization will have a huge impact on the overall performance of the algorithm.

Algorithm 6 Proposed Algorithm: Finding shortest path in presence of blocking nodes - Optimized Version

1: **procedure FindingShortestPathOptimized**(X, n) ▷ X: set of blocking nodes, n:dimension of cube
2: $P \leftarrow []$ ▷ Start with an empty ordering
3: $N \leftarrow \{1, 2, \ldots, n\}$
4: $G \leftarrow CreateGraph(X)$
5: **for** $i = 1 \rightarrow |N|$ **do**
6: $\alpha \leftarrow N_i$ ▷ Get i-th element of N
7: $P \leftarrow P + [\alpha]$ ▷ Append α to the end of P
8: $G' \leftarrow G$ ▷ Keep a copy before removing nodes
9: **for** $j = 1 \rightarrow |X|$ **do**
10: **if** $\{P\} \not\subseteq X_i$ **then**
11: remove X_i from G
12: **end if**
13: **end for**
14: **if** InclusionExclusionCounting$(G, n, P) > 0$ **then**
15: $N \leftarrow N \setminus \{\alpha\}$ ▷ Remove α from set N
16: $i \leftarrow 1$ ▷ Reset to beginning of N
17: **else**
18: $P \leftarrow P \setminus [\alpha]$ ▷ Remove α from the end of P
19: $G \leftarrow G'$ ▷ Roll back to unchanged graph
20: **end if**
21: **if** $\alpha == n$ and $P == []$ **then** ▷ Reached n but no first element yet
22: break ▷ There is no answer
23: **end if**
24: **end for**
25: return P ▷ In case of no answer, P will be empty
26: **end procedure**

3.3.9 Counting All Possible Shortest Paths

With the proposed algorithm, not only can we find a shortest path but also we can count all possible shortest paths.

To do so, we can simply try all members of N as the first element of P and sum up the result of inclusionExclusionCounting each time. In other words, we count the number of valid permutations for all $P = [\alpha], \alpha \in N = \{1, 2, \ldots, n\}$ and sum them up.

Algorithm 7 Finding the total number of shortest path in presence of blocking nodes

```
1: procedure CountingAllShortestPaths(X, n)        ▷ X: set of blocking nodes,
   n:dimension of cube
2:     N ← {1, 2, ..., n}
3:     G ← CreateGraph(X)
4:     total ← 0
5:     for i = 1 → |N| do
6:         α ← N_i
7:         P ← [α]
8:         total ← total+ InclusionExclusionCounting (G, n, P)
9:     end for
10:    return total
11: end procedure
```

3.3.10 Examples

To elaborate each step of the algorithm, we provide some examples in this section.

Example 3.10. In a 5-cube:
$m = 7$
$X_1 = \{2, 3\}$
$X_2 = \{1, 2, 3\}$
$X_3 = \{1, 2, 5\}$
$X_4 = \{1, 2, 3, 4\}$
$X_5 = \{1, 2, 4, 5\}$
$X_6 = \{1, 2, 3, 5\}$
$X_7 = \{1, 5\}$

The 5-cube has been depicted in Figure 3.5. As we discussed in Chapter 1, an n dimensional hypercube can be decomposed into lower dimensional hypercubes. Here we

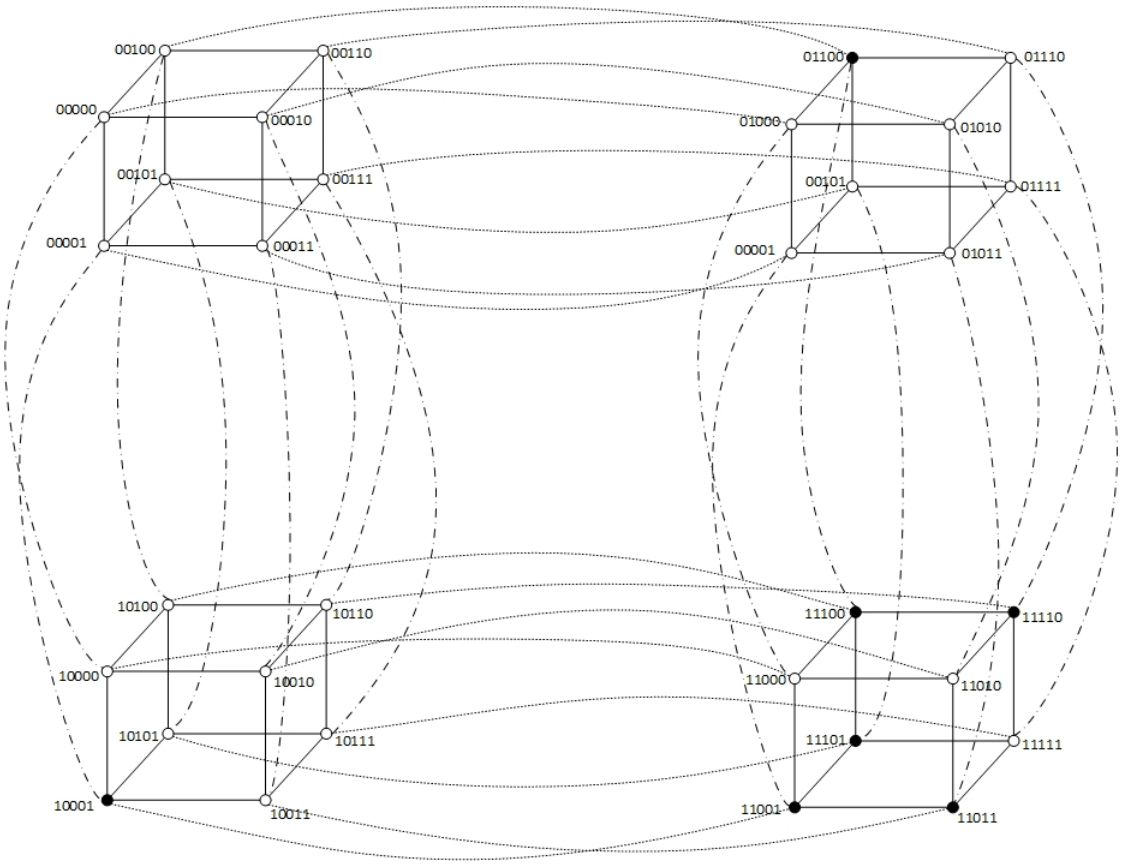

Figure 3.5: 5-cube of Example 3.10

decomposed a 5−cube into 4 3−cubes. The DAG of the blocking nodes is illustrated in Figure 3.6.

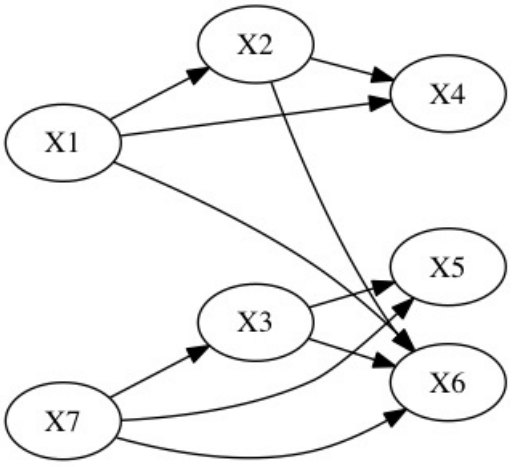

Figure 3.6: DAG of Example 3.10

At first, 1 will be chosen as the first element of P and then the dynamic program-

ming table will be calculated based upon that.

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	2	0	0	0	0
X_3	2	1	0	0	0
X_4	6	2	0	0	0
X_5	6	4	1	0	0
X_6	6	6	1	0	0
X_7	1	0	0	0	0

Table 3.1: DP table of Example 3.10, $P = [1], total = 4$

In Table 3.1:

- The first row is completely zero because $\{P\} \not\subseteq X_1$.
- Columns 4 and 5 are zero since there is no path of size 4 or 5.
- All columns of X_7 are zero but the first one. This is because there is no path of size more than one that ends at X_7.
- As an example $T(X_5, 2) = T(X_3, 1) * (|X_5| - |X_3|)! + T(X_7, 1) * (|X_5| - |X_7|)! = 2 * 1! + 1 * 2! = 4$.

The inclusion-exclusion formula will be calculated based on the DP table as follows:

The initial value will be:

$$total = (n - |P|)! = (5 - 1)! = 24$$

And then in the loop of Algorithm 3:

$$
\begin{aligned}
total =& total \\
& - \sum_{i_1=1}^{7} |Y_{i_1}| \\
& + \sum_{1 \le i_1 < i_2 \le 7} |Y_{i_1} \cap Y_{i_2}| \\
& - \sum_{1 \le i_1 < i_2 < i_3 \le 7} |Y_{i_1} \cap Y_{i_2} \cap Y_{i_3}| \\
& + \sum_{1 \le i_1 < i_2 < i_3 < i_4 \le 7} |Y_{i_1} \cap Y_{i_2} \cap Y_{i_3} \cap Y_{i_4}| \\
& - \sum_{1 \le i_1 < i_2 < i_3 < i_4 < i_5 \le 7} |Y_{i_1} \cap Y_{i_2} \cap Y_{i_3} \cap Y_{i_4} \cap Y_{i_5}| \\
& = total \\
& - \sum_{i=1}^{7} (T(X_i, 1) * (5 - |X_i|)!) \\
& + \sum_{i=1}^{7} (T(X_i, 2) * (5 - |X_i|)!) \\
& - \sum_{i=1}^{7} (T(X_i, 3) * (5 - |X_i|)!) \\
& + \sum_{i=1}^{7} (T(X_i, 4) * (5 - |X_i|)!) \\
& - \sum_{i=1}^{7} (T(X_i, 5) * (5 - |X_i|)!) \\
& = 24 \\
& - (0 * 3! + 2 * 2! + 2 * 2! + 6 * 1! + 6 * 1! + 6 * 1! + 1 * 4!) \\
& + (0 * 3! + 0 * 2! + 1 * 2! + 2 * 1! + 4 * 1! + 6 * 1! + 0 * 4!) \\
& - (0 * 3! + 0 * 2! + 0 * 2! + 1 * 1! + 1 * 1! + 0 * 1! + 0 * 4!) \\
& + (0 * 0! + 0 * 2! + 0 * 2! + 0 * 1! + 0 * 1! + 0 * 1! + 0 * 4!) \\
& - (0 * 0! + 0 * 2! + 0 * 2! + 0 * 1! + 0 * 1! + 0 * 1! + 0 * 4!) \\
& = 24 - 30 + 14 - 0 + 0 \\
& = 4
\end{aligned}
$$

Therefore, if we choose 1 as the first element of P, there will be 4 different permutations starting with 1 and satisfying the validation condition and thus making a

valid path.

Therefore, 1 will be removed from the set N and we try number 2 as the second element. Here is the DP table for $P = [1, 2]$.

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	1	0	0	0	0
X_3	1	0	0	0	0
X_4	2	1	0	0	0
X_5	2	1	0	0	0
X_6	2	2	0	0	0
X_7	0	0	0	0	0

Table 3.2: DP table of Example 3.10, $P = [1, 2], total = 0$

And the *total* will be:

$$\begin{aligned}
total &= 6 \\
&\quad - (0 * 3! + 1 * 2! + 1 * 2! + 2 * 1! + 2 * 1! + 2 * 1! + 0 * 4!) \\
&\quad + (0 * 3! + 0 * 2! + 0 * 2! + 1 * 1! + 1 * 1! + 2 * 1! + 0 * 4!) \\
&\quad - (0) \\
&\quad + (0) \\
&\quad - (0) \\
&\quad = 6 - 10 + 4 - 0 + 0 \\
&\quad = 0
\end{aligned}$$

Since *total* is zero, inclusionExclusionCounting() function returns 0 and then we need to go to step 11 of the main algorithm. As a result, 2 will be put back to the set N and we continue trying the next element which is 3 as the second element.

As can be seen in Table 3.3, X_3's row is now all zero since P is not a subset of X_3 anymore. And therefore there is not any path of length 3 or more.

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	1	0	0	0	0
X_3	0	0	0	0	0
X_4	2	1	0	0	0
X_5	0	0	0	0	0
X_6	2	1	0	0	0
X_7	0	0	0	0	0

Table 3.3: DP table of Example 3.10, $P = [1, 3], total = 2$

And the *total* will be:

$$\begin{aligned} total &= (5-2)! \\ &\quad - (0*3! + 1*2! + 0*2! + 2*1! + 0*1! + 2*1! + 0*4!) \\ &\quad + (0*3! + 0*2! + 0*2! + 1*1! + 0*1! + 1*1! + 0*4!) \\ &\quad - (0) \\ &\quad + (0) \\ &\quad - (0) \\ &\quad = 6 - 6 + 2 - 0 + 0 \\ &\quad = 2 \end{aligned}$$

Therefore, we can continue with number 3 as the second element.

The next choice will be adding 2 and make $P = [1, 3, 2]$, but since we already have $X_2 = \{1, 2, 3\}$ therefore $X_2 = \{P\}$ which will be detected in Line 3 of inclusionExclusionCounting() function (Algorithm 4). The same will happen when the algorithm tries $P = [1, 4, 3, 2]$ which makes its set equal to X_4.

The rest of the example is shown by the tables and the total value at each step without further explanations.

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	0	0	0	0	0
X_4	1	0	0	0	0
X_5	0	0	0	0	0
X_6	1	0	0	0	0
X_7	0	0	0	0	0

Table 3.4: DP table of Example 3.10, $P = [1, 3, 2], total = 0$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	0	0	0	0	0
X_4	1	0	0	0	0
X_5	0	0	0	0	0
X_6	0	0	0	0	0
X_7	0	0	0	0	0

Table 3.5: DP table of Example 3.10, $P = [1, 3, 4], total = 1$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	0	0	0	0	0
X_4	0	0	0	0	0
X_5	0	0	0	0	0
X_6	0	0	0	0	0
X_7	0	0	0	0	0

Table 3.6: DP table of Example 3.10, $P = [1, 3, 4, 5], total = 1$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	0	0	0	0	0
X_4	0	0	0	0	0
X_5	0	0	0	0	0
X_6	0	0	0	0	0
X_7	0	0	0	0	0

Table 3.7: DP table of Example 3.10, $P = [1, 3, 4, 5, 2], total = 1$

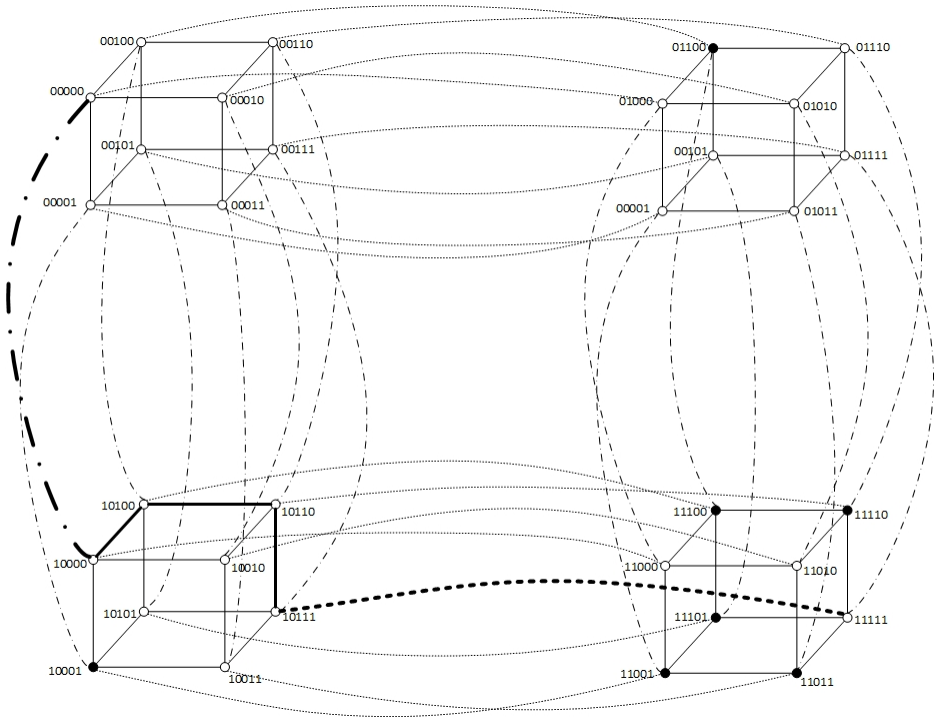

Figure 3.7: The solution of Example 3.10 that is found by the algorithm is depicted in black

As can be seen from the above tables, as P grows, the number of cells that are zero increases. The reason is simply because P will not be a subset of many of the blocking nodes. Finally the answer is $P = [1, 3, 4, 5, 2]$ which is equal to the path $[00000 - 10000 - 10100 - 10110 - 11101 - 11111]$ which is depicted in Figure 3.7.

Now we add some other blocking nodes to this example to make it impossible to find the shortest path.

Example 3.11. In a 5-cube depicted in Figure 3.8:
$m = 9$
$X_1 = \{2, 3\}$
$X_2 = \{1, 2, 3\}$
$X_3 = \{1, 2, 5\}$
$X_4 = \{1, 2, 3, 4\}$
$X_5 = \{1, 2, 4, 5\}$
$X_6 = \{1, 2, 3, 5\}$
$X_7 = \{1, 3, 4\}$
$X_8 = \{1, 3, 4, 5\}$
$X_9 = \{2, 3, 4, 5\}$

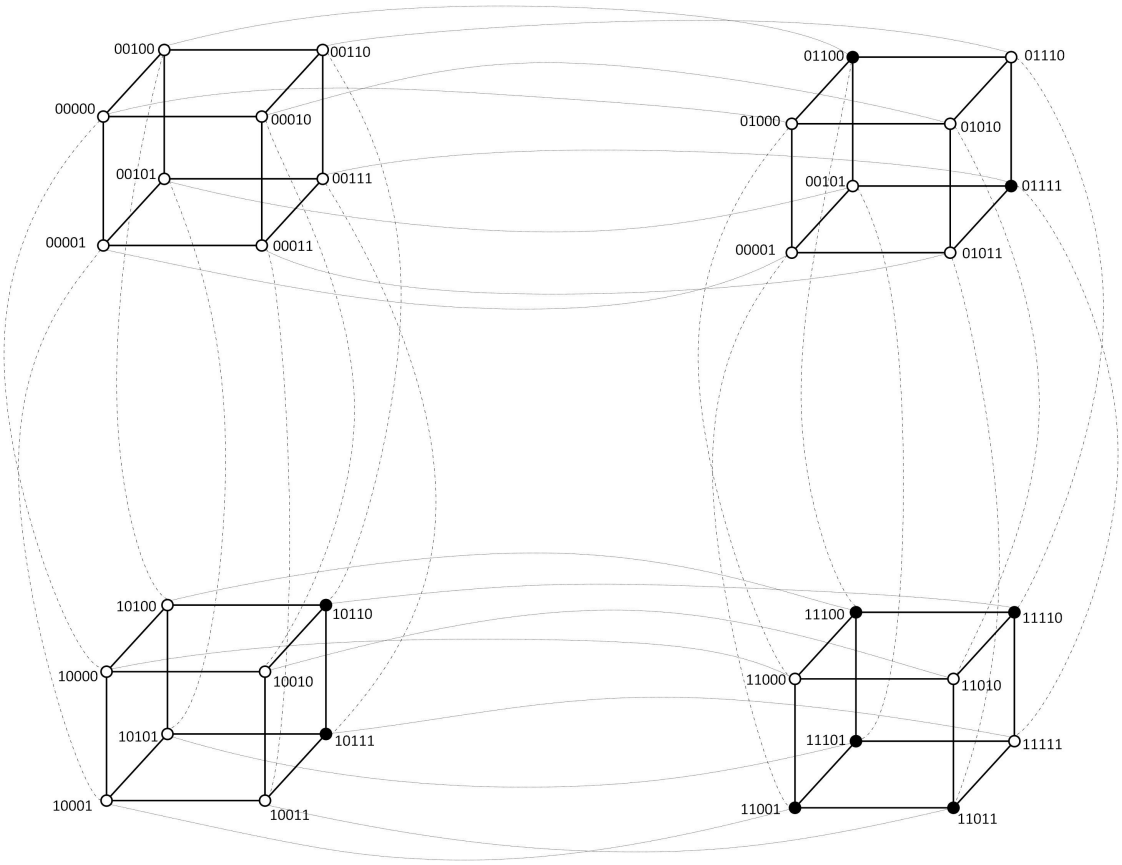

Figure 3.8: 5-cube of Example 3.11

Our algorithm tries all elements as the first element but the total values for all of them is zero and therefore there is no answer for this example.

	1	2	3	4	5
X_1	1	0	0	0	0
X_2	2	1	0	0	0
X_3	2	1	0	0	0
X_4	6	6	1	0	0
X_5	6	4	1	0	0
X_6	6	6	2	0	0
X_7	2	0	0	0	0
X_8	6	2	0	0	0
X_9	0	0	0	0	0

Table 3.8: DP table of Example 3.11, $P = [1], total = 0$

	1	2	3	4	5
X_1	1	0	0	0	0
X_2	2	1	0	0	0
X_3	2	1	0	0	0
X_4	6	4	1	0	0
X_5	6	4	1	0	0
X_6	6	6	2	0	0
X_7	0	0	0	0	0
X_8	0	0	0	0	0
X_9	6	0	0	0	0

Table 3.9: DP table of Example 3.11, $P = [2], total = 0$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	2	0	0	0	0
X_3	0	0	0	0	0
X_4	6	4	0	0	0
X_5	0	0	0	0	0
X_6	6	2	0	0	0
X_7	2	0	0	0	0
X_8	6	2	0	0	0
X_9	6	0	0	0	0

Table 3.10: DP table of Example 3.11, $P = [3], total = 0$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	0	0	0	0	0
X_4	6	2	0	0	0
X_5	6	0	0	0	0
X_6	0	0	0	0	0
X_7	2	0	0	0	0
X_8	6	2	0	0	0
X_9	6	0	0	0	0

Table 3.11: DP table of Example 3.11, $P = [4], total = 0$

	1	2	3	4	5
X_1	0	0	0	0	0
X_2	0	0	0	0	0
X_3	2	0	0	0	0
X_4	0	0	0	0	0
X_5	6	2	0	0	0
X_6	6	2	0	0	0
X_7	0	0	0	0	0
X_8	6	0	0	0	0
X_9	6	0	0	0	0

Table 3.12: DP table of Example 3.11, $P = [5], total = 0$

Chapter 4

Conclusion

In this thesis we proposed a polynomial algorithm for a routing paradigm in the hypercube, one of the most popular types of interconnection networks. The paradigm was investigating the existence of shortest path between two given nodes in a faulty hypercube, i.e. when some nodes are faulty and cannot be used for routing. We reduced the problem to a permutation problem and we used inclusion-exclusion principle and dynamic programming technique to solve the problem efficiently. Our algorithm was constructive and thus it can construct the shortest path if it exists. The proposed algorithm was based on the fact that the state of all nodes (whether they are faulty or not) is known to the source node and hence the source node can check whether the shortest path exists or not before sending its data. We are working to extend our algorithms to other cube-like networks. As discussed in Chapter 1, due to the specific similarity of augmented cube and folded cube to hypercube, we believe that these are two proper candidates to start our investigation with. The idea also can be investigated to other symmetric networks and to other routing paradigms as well.

Currently, our algorithm has time complexity of $O(n^2m^2)$ (n as dimension of hypercube and m number of faulty nodes) that we believe can be improved in future work. One observation to do so is that the set of edges in the faulty nodes graph is a transitive set. That implies if node u is connected to node v and node v is connected to node t, thus there should be an edge from u to t. This makes the graph a dense graph with many edges. We believe as a future work, we can take advantage of transitivity property and make the graph a sparse graph and this helps to improve the time complexity of dynamic programming phase of the proposed algorithm.

Bibliography

[1] Selim G. Akl. *Parallel computation: Models and methods*. Prentice Hall Inc., 1997.

[2] Eddie Cheng, Shuhong Gao, Ke Qiu, and Zhizhang Shen. On disjoint shortest paths routing on the hypercube. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications*, pages 375–383. Springer-Verlag, 2009.

[3] Eddie Cheng, Keivan Noroozi, Ke Qiu, and Zhizhang Shen. Routing on the hypercube with blocking/faulty nodes. Unpublished paper.

[4] Ge-Ming Chiu and Shui-Pao Wu. A fault-tolerant routing strategy in hypercube multiprocessors. *Computers, IEEE Transaction on*, 45(2), 1996.

[5] Gi-Ming Chiu and Shui-Pao Wu. A fault-tolerant routing strategy in hypercube multicomputers. *Computers, IEEE Transaction on*, 45(2), 1996.

[6] S. A. Choudum and V. Sunitha. Augmented cubes. In *Networks*, volume 40, pages 70–84. Wiley Periodicals, 2002.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] Ahmed El-Amawy and Shahram Latifi. Properties and performance of folded hypercubes. *Parallel and Distributed Systems, IEEE Transaction on*, 2(1), 1991.

[9] Shuhong Gao, Beth Novick, and Ke Qiu. From hall's matching theorem to optimal routing on hypercubes. *Journal of Combinatorial Theory, Series B*, 74(2):291 – 301, 1998.

[10] Wen-Jing Hsu. Fibonacci cubes - a new interconnection topology. *Parallel and Distributed Systems, IEEE Transaction on*, 4(1), 1993.

[11] Michael Jurczyk, Howard Jay Siegel, and Craig Stunkel. *Interconnection networks for parallel computers*. John Wiley & Sons, Inc., 2007.

[12] Keiichi Kaneko and Hideo Ito. Fault-tolerant routing algorithms for hypercube networks. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 218–224. IEEE Computer Society, 1999.

[13] Tze Chiang Lee and John P. Hayes. A fault-tolerant communication scheme for hypercube computers. *Computers, IEEE Transaction on*, 41(10), 1992.

[14] Yamin Li, Shietung Peng, and Wanming Chu. Efficient collective communications in dual-cube. *The Journal of Supercomputing*, 28(1):71–90, 2004.

[15] Peter K. K. Loh, Wen Jing Hsu, and Yi Pan. The exchanged hypercube. *Parallel and Distributed Systems, IEEE Transaction on*, 16(9), 2005.

[16] Franco P. Preparata and Jean Vuillemin. The cube-connected-cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.

[17] Deqiang Wang and Lianchang Zhao. The twisted-cube connected networks. *Journal of Computer Science and Technology*, 14(2), 1999.